Ground Software Technologies – Embracing Change: Mission Drivers and Technology Opportunities to Enable Long Lived Missions

Brian J. Giovannoni[1]
Jet Propulsion Laboratory, California Institute of Technology, Pasadena, California 91109

Mission lifecycles have proven to extend well beyond their original design. The benefits to this are countless but introduce challenges in today's rapidly changing ground infrastructure and software technologies used to enable mission success. What remains constant is the risk posture missions maintain when accepting change and the use of new technologies. Larger missions are ready for change in early lifecycle development but near launch and especially in operations, few continue to evolve beyond what is set in place in phase C.

This paper will discuss how the Advance Multi-Mission Operations System (AMMOS) intends to address, three driving missions concerns: Maintaining functionality (hardware/software) for decades, rapidly responding to security vulnerabilities in software, and finally the ability to quickly evolve infrastructure and software changes.

These driving concerns are briefly described below:

1. Maintaining functionality (hardware/software) for decades

Hardware updates considerably faster than 10 years ago. Expectations that a system can remain in place for more than 10 years is no longer valid. Expecting to find hardware replacements for a system older than 5 years will increasingly become more and more challenging. How than do missions plan for hardware changes for long lived missions?

Principle Objective: Provide abstraction by virtualizing and containerizing software abstract away any hardware dependencies and package up the application lightweight units.

2. Rapidly responding to security vulnerabilities in software

Cost is often the main impediment and largely driven by the revalidation and testing of system that undergo change. In todays, environment security updates are a major diver demanding systems remain up to date. How then do missions accept these changes and avoid large testing efforts?

Principle Objective: Help reduce the cost of re-testing by automation of testing, deployment, and compartmentalizing change.

[1] Manager, Customer Interface Management Office, Interplanetary Network Directorate, M/S 301-350B.

3. Ability to quickly evolve infrastructure and software changes

Responding quickly to change is similar to the second concern in this paper regarding security vulnerabilities. In this case, it address broader concerns of updating software and infrastructure on a more realistic timeline. How do missions stay up to date with the most recent versions of software and allowing for improved functionality?

Principle Objective: Use continuous integration techniques at the system level to ensure rapid turnaround.

This paper explores each of these concerns in more detail. It focuses the AMMOS's current plans, challenges and current roadmap.

I. Introduction

THIS paper discusses modern approaches to address mission system longevity. Gone are the days when mission operation systems can remain stagnate. Ninety-day missions turn into ten-year missions, yet planning for change is often handled ad hoc. Computer hardware changes faster than ever, software seemingly requires patches every month, closed and walled off systems are considerably more exposed. The legacy approach to system maintenance, "build and do not touch," is no longer tenable.

Designing for extended "extended" missions needs to become the norm. To do this it is important to ensure the appropriate level of system maintenance in terms of budget and schedule is planned. Maintenance planning is needed to support updating system hardware and software as well as to support the testing required for these changes (See AIAA SpaceOps 2016 paper: Hidden Costs of Unsupported Software, Obsolescence and Non-Standards; The Importance and Value of a Multi-Mission Software Program[1]). Acknowledging the cost of maintenance for the entire lifecycle of the mission, including planning and staffing support, will allow this same support to continue in extended mission.

II. Primer – Containerization and Continuous Integration

This section is intended to provide just enough information to understand the technologies and concepts used throughout the paper. It briefly describes containerization, continuous integration, and the technology benefits for mission system development and deployment.

A. Containerization – What is it and why does it matter?

Without getting into too much detail, let us briefly discuss what a container is. A container is a stand-alone, executable image bundling software (an application) and everything needed for it to run. This includes runtime executables, required system tools (3rd party), required libraries, and configuration settings. The work described in this paper is based on the Docker[2] container.

Container technology brings numerous benefits to a mission system as outlined in Table 1.

Attribute	Mission System Benefits
1. Portable deployments	• Containers run on multiple platforms (Windows, MacOS, Linux,

[2] Docker, "What is Docker" [online reference], URL: https://www.docker.com/what-docker

	Cloud) allowing for bundling and deployment of mission system applications.
2. Application-centric	• Containers deploy and deliver applications enabling system teams to focus on mission operations capabilities. Discrete bundles tease apart complex dependencies.
3. Support for automatic container builds	• Enables process automation: code change, automatic build and unit test, system deployment and automatic system test.
4. Discrete self-sufficient bundles	• Simplifies adoption. • Simplifies deployment. • Supports maintaining functionality by removing hardware infrastructure and environment dependencies.

Table 1. Containerized software mission system benefits

B. Continuous Integration – What is it and why does it matter?

Continuous Integration (CI) is a derivative of agile software development practices in which developers continually check in code for a nightly build process. Builds are regularly run against automated regression testing and integration problems addressed very frequently. This has proven successful for application development. This paper takes the concept of continuous integration for application development and extends it to include system integration.

The benefits continuous integration bring to a mission system are enumerated in Table 2.

Attribute	Mission System Benefits
2. Reduces risk of integration	• Frequent integration facilitates planning and scheduling.
3. System and 3rd party patching - smoke testing	• Determines if system and third party patches break the system.
5. Support smaller team (Integration / Test / Deployment)	• Stay on top of needed updates avoiding obsolescence and security vulnerabilities even with a small staff.

Table 2. Continuous integration mission system benefits

C. Important Technologies

Docker and a Continuous Integration process require support technologies to enable a structured workflow and automation. This section highlights the important technologies AMMOS is using in along with references.

Technology	Reference
Artifactory[3]	A development tool that supports binary management, works with different software package management systems, and easily integrates into a continuous integration workflow.
YUM – Yellowdog Updater Modified[4]	A package installer/remover used for Redhat Package Managed systems.
Jenkins[5]	An open source automation server that supports building, deploying and automating development projects.

[3] Artifactory, "JFrog Artifactory Enterprise Universal Artifact Manager " [Vendor Website], URL: https://jfrog.com/artifactory/
[4] YUM, "Yellowdog Updater Modified" [Open Source Collaboration Website], URL: http://yum.baseurl.org/
[5] Jenkins, "Open source automation server" [Website], URL: https://jenkins.io/

Software Repositories	AMMOS software repositories used: GIT / GITHUB SVN CVS

Table 3. Important technology references

III. Extending the Lifetime of Functionality

Gone are the days when systems can remain untouched and expected to remain in a working state year after year. Hardware fails and exact match replacements are more difficult to come by as infrastructure changes each year. Software ages more quickly with changes in operating systems and updates in 3rd party dependencies. These dependencies can dictate mandatory change due to security vulnerabilities. The section discusses several approaches that capitalize on containerization technology in order to lengthen the lifespan of software functionality. The ability to run anywhere increases application lifespan. In order to do this you need to isolate applications, understand system dependencies, and ensure that best efforts are made to make applications host agnostic.

A. Application Isolation

A Docker container uses abstraction to isolate software. The runtime environment is consistent on multiple platforms. Multiple environments can run and remain isolated on a single machine. Docker containers run on Linux distributions, Windows, VMs, bare-metal and in the cloud. Based on open standards there is an ever-increasing open source community contributing and maintain Docker containers. Dockerizing an application allows for a great degree of portability and possibly an increase in lifespan. There are never guarantees but if Docker container technology stays in vogue, it can prolong the life of the application.

Docker containers isolate applications from other applications as well as the infrastructure. If done correctly, this strong isolation tightens security. This allows older applications to run in sandboxed configurations. If a mission cannot update the application, at least it is possible to wall it off.

B. System Dependencies – Understanding Application Updates

Docker images give you complete control over the contents. It may not seem obvious but understanding all your application dependencies and having a way to control the contents of a container can support application maintenance. To the largest extent possible, breaking the system into compartmentalized objects allows for planned change. Applications that can be will be regularly updated. Applications that have less support can be isolated.

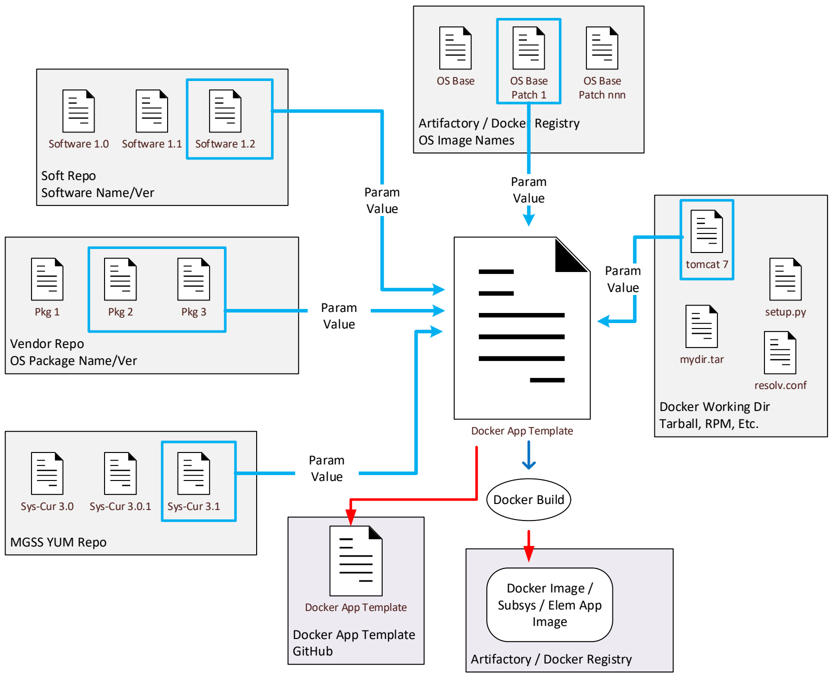


Fig 1. Application Templates and supporting infrastructure

Figure 1. Illustrates the use of a version-controlled template defining applications content, version controlled 3rd party and operating system packages, version controlled system cores and patches, and finally repositories for the resulting builds.

C. Agnostic Containers – Portable Deployments

Applications are often developed with dependencies tied to a machine's specific configuration: networking, storage, logging, etc. A Docker container allows for bundling of an application and all its dependencies into a single object. In addition, Docker supports abstraction for machine-specific settings. When this abstraction is used, containers can run - unchanged - on many different machines, with different configurations.

IV. Reducing the Cost of Testing and Deployment

There is no changing the fact that budgets in operational phases of a mission will decline. Once a mission is extended, budgets are reduced even more. This section defines an approach that uses continuous integration to enable more frequent testing in operations with smaller integration and test teams.

The workflow is as follows: patch, build application, test application, system integration, system test and report. Patching is covered in more detail later in section V.

A. Application Build and Test

Continuous integration must provide rapid feedback to developers, system integrators and users. However, to do this you need to establish a fast and efficient process. Automation pipelines are used that build, test and report results. For the best results, the test environment should mimic the production environment. Using Docker containers in the test and operations venues yields the best results. Containerizing 3rd party components like database software, and operating system patches are included in the containers or built as microservices used by applications deployed in containers (See Fig 1.).

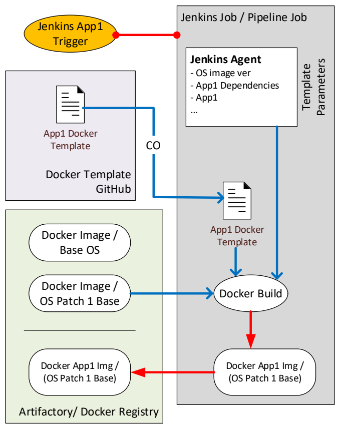


Fig 2. Application Image Build Workflow

Figure 2. Illustrates how the pipeline builds applications using controlled templates. Templates are stored in an accessible repository and can be used in other workflows. The test pipeline uses a suite of automated tests able to check a large part of the code base for bugs. The tests are kicked off from a simple command and integrated into the pipeline. The result of running the test suite indicate pass and failed conditions. Failure of a test is reported and the resulting application is removed from system test (see Fig 3.).

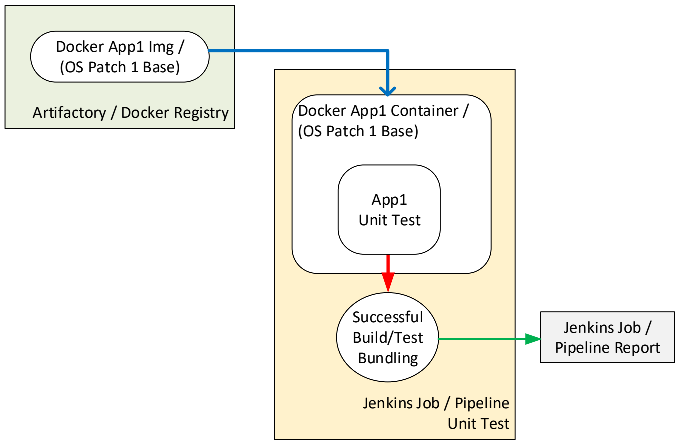
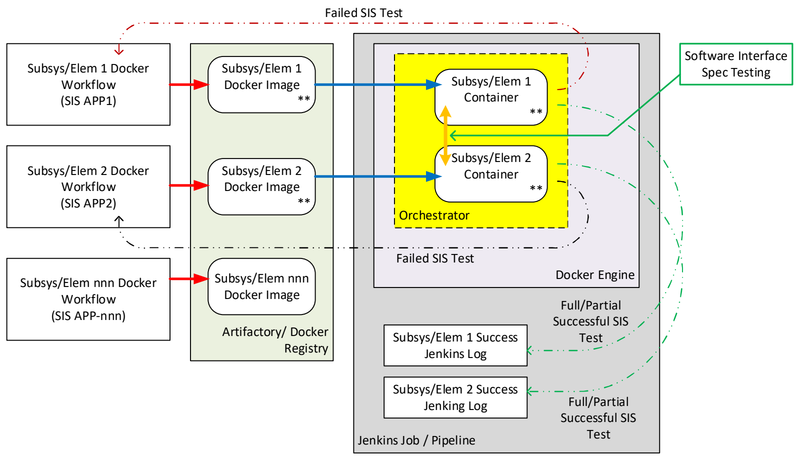

Docker App1 Img / (OS Patch 1 Base)
Artifactory / Docker Registry
Docker App1 Container / (OS Patch 1 Base)
App1 Unit Test
Successful Build/Test Bundling
Jenkins Job / Pipeline Report
Jenkins Job / Pipeline Unit Test

Fig 3. Application Image Test Workflow

A. Automating System Test

The application CI approach defined in the previous section improves testing of a mission system. AMMOS development teams are expected to use CI. These applications are tested in the context of a deployed mission operation system. Extending CI at the system level allows missions to take advantage of detailed unit testing provided by the applications. System testing adds: system interface testing, and system specific non-functional and end-to-end testing.

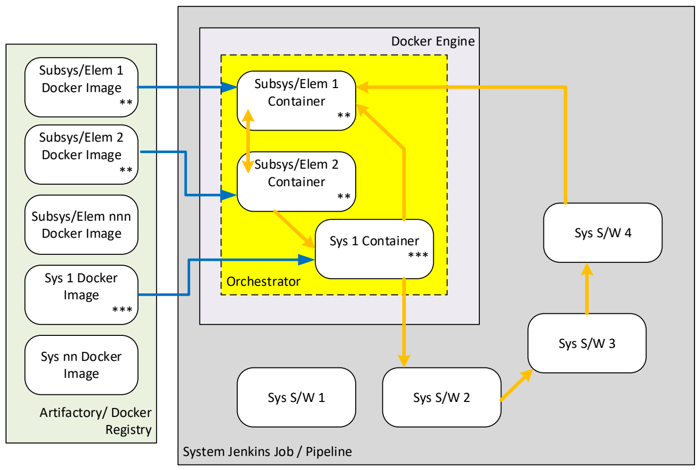
Figure 4. Illustrates how system testing can rapidly verify key system interfaces. Automating system interface testing allows individual applications to change without the need to perform a complete system test cycle.



Failed SIS Test
Subsys/Elem 1 Docker Workflow (SIS APP1)
Subsys/Elem 2 Docker Workflow (SIS APP2)
Subsys/Elem nnn Docker Workflow (SIS APP-nnn)
Subsys/Elem 1 Docker Image **
Subsys/Elem 2 Docker Image **
Subsys/Elem nnn Docker Image
Artifactory/ Docker Registry
Subsys/Elem 1 Container **
Subsys/Elem 2 Container **
Orchestrator
Failed SIS Test
Docker Engine
Software Interface Spec Testing
Full/Partial Successful SIS Test
Subsys/Elem 1 Success Jenkins Log
Subsys/Elem 2 Success Jenking Log
Full/Partial Successful SIS Test
Jenkins Job / Pipeline

Fig 4. Automated System Interface Test Workflow

System tests are often non-functional in nature dealing with performance, throughput and security, but can also include end-to-end tests e.g. the production of mission products involving multiple applications.



Docker Engine
Subsys/Elem 1 Docker Image **
Subsys/Elem 2 Docker Image **
Subsys/Elem nnn Docker Image
Sys 1 Docker Image ***
Sys nn Docker Image
Artifactory/ Docker Registry
Subsys/Elem 1 Container **
Subsys/Elem 2 Container **
Sys 1 Container ***
Orchestrator
Sys S/W 4
Sys S/W 3
Sys S/W 1
Sys S/W 2
System Jenkins Job / Pipeline

Fig 5. Automated System Test Workflow

It is possible to use a configurable combination of automated tests or simply verify the update of a single application running in as part of the system. To test a single application the workflow need only include the applications system interface test and end-to-end tests required to ensure functionality meets requirements. Application deployment is automated, the test cases are automated and gone is the need for three months of manual and comprehensive system testing.

V. Responding to Rapid Change – Operating System and Software Patching

Section IV discussed how a mission could use system interface and end-to-end automated tests to verify updates. This section goes into detail regarding changes to the operating system and 3rd party software. Automation can reduce the effort required to determine if operating systems and support software break mission system functionality. For a small change, a traditional system test cycle is not needed. By using the application and system pipelines, we allow the mission to selectively compose levels of test. Application failing test after an update need only be rolled back. Additionally, rolling back to older versions of working states is also automatable.

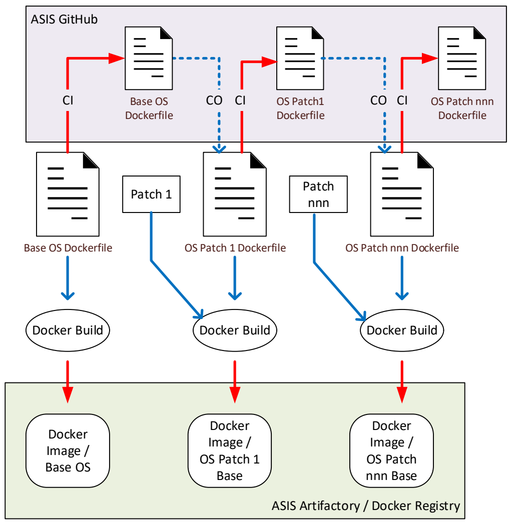


Fig 6. Controlled Operating System Updates

Integrating CI pipelines into each application build allows the system to build container updates without an entire application development lifecycle. Application automated tests generate success criteria and feed the system interface and end-to-end testing. Application need not update functionality just be retested at the system level with patches. Important security patches can be integrated into the system without perturbing application development cycles.

VI. Conclusion

Designing for extended "extended" missions needs to become the norm. The process depends on ensuring the appropriate level of system maintenance planning. Modularizing using containers and instituting automation allow for smaller teams. The process for build, test, deploy, integrate and test again should using CI early in mission life cycles then become established and considered the norm in operations phase.

Acronyms and Abbreviations

CI	Continuous Integration
MGSS	Multimission Ground System and Services
OS	Operating System
RPM	Redhat Package Manager

Acknowledgements

Thank you to the following people who have contributed ideas and produced image content: The MGSS System Engineering Team at JPL most notably: Vu Nguyen, Nik Sheurpukdi. Additional thanks to Jeff Estefan, Michelle Thompson MD for proofing and support.

This paper was developed at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

References

[1]Giovannoni, Boyles, "Hidden Costs of Unsupported Software, Obsolescence and Non-Standards; The Importance and Value of a Multi-Mission Software Program", AIAA SpaceOps 2016.